On-board image compression for the HST Advanced Camera for Surveys

Richard L. White[a] and Ira Becker[b]

[a]Space Telescope Science Institute, 3700 San Martin Drive, Baltimore, MD 21218

[b]Ball Aerospace, P.O. Box 1062, MS FA-3, Boulder, CO 80306

ABSTRACT

The Wide Field Camera in the Advanced Camera for Surveys (ACS) produces very large 4096×4096 pixel images. We will have on-board image compression in order to reduce both the storage requirements at the telescope and the time to transmit the data to the ground. This is the first time on-board compression has been included in an Hubble Space Telescope instrument.

We have developed a new lossless image compression algorithm that is designed to compress the CCD data by factors of 2 to 3.5 with the minimum possible computational load on the ACS computer. The new algorithm takes differences of adjacent pixels and then compresses the difference in pairs, producing output codes of 1, 2, or 4 bytes for each pair. The pair-coding algorithm gives slightly inferior compression to the Rice algorithm (which was our method of choice) but is more than three times faster than Rice on our computer. The Rice algorithm was unfortunately too slow for us to use, but the pair-coding algorithm is fast enough to handle high data rates even on our 16-MHz 80386 computer.

This paper will describe the details of the compression algorithm and its implementation in the ACS flight software. Important implementation problems include the unpredictable data volume after compression and the need to compress four independent data streams during readout from the Wide Field Camera CCDs. We will also describe the compression performance of the new algorithm on various types of astronomical images.

Keywords: Hubble, HST Advanced Camera, CCDs, image compression, pair-coding

1. INTRODUCTION

The Hubble Space Telescope Advanced Camera for Surveys (HST ACS) will substantially improve the sensitivity, sky coverage, and resolution of HST's current complement of cameras. (See the paper by Ford *et al.*[1] in these proceedings for details of the ACS design and performance.) The largest format camera in the ACS is the Wide Field Camera (WFC), which has a pair of 2048×4096 CCD detectors with 0.05" pixels. The field of view of this camera, 200×200", is both substantially larger and much better sampled than the field of the HST Wide-Field Planetary Camera (WFPC2).

The cost of this increase in resolution and field of view is a large increase in the volume of data produced by the camera. The WFPC2 produces images with 1600×1600 2-byte pixels, for a data volume of ~ 5 Mbytes per image. The ACS WFC images are more than 6 times larger, with a data volume of 32 Mbytes/image. (Note that the WFC images have some additional overscan-pixels at the ends of each row. For purposes of simplicity in this paper, the overscan pixels are ignored.)

The HST data follow a path with several bottlenecks that strictly limit the total volume of data that can be transmitted to the ground. First, the data channel between the science instruments and the spacecraft computer has a bandwidth of 0.125 Mbytes/sec; this limits the rate at which a large image can be read out of the ACS for storage on-board. It requires 256 sec to read out an ACS WFC image through this channel. Second, the ground link from HST also has a maximum bandwidth of 0.125 Mbytes/sec; moreover, this link is shared by many spacecraft (including the space shuttle) and so is available to HST for only of order an hour per day. This limits the total number of images that can be acquired each day.

Data compression allows us to reduce the data volume required to transmit an ACS WFC image. Most pixel values in astronomical images fall in a relatively narrow range near the mean sky brightness level; also, pixel values

Other author information – Email: rlw@stsci.edu, ibecker@ball.com

Table 1. Acronyms

A/D	Analog-to-Digital
ACS	Advanced Camera for Surveys
CCD	Charge-Coupled Device
FIFO	First-In First-Out (Device)
FSW	Flight Software
HST	Hubble Space Telescope
WFC	Wide-Field Channel
WFPC2	Wide-Field and Planetary Camera 2

are usually closely correlated with the brightnesses of neighboring pixels. Lossless image compression algorithms can reduce the data size of ACS images by a factor of 2–4, from 2 bytes (16 bits) per pixel to only 4–8 bits per pixel. By compressing the ACS WFC images using the ACS on-board computer, we are able to reduce both the time to read out the data (making it possible to take more closely spaced images) and the time to transmit the data to the ground (making it possible to acquire more images each day.) On-board compression thus plays an important role in enabling more and better science to be carried out with the ACS.

In this paper, §2 discusses the requirements that affected the design of the compression method; §3 describes the mathematical algorithm we have developed for ACS image compression; §4 discusses the detailed implementation of the algorithm (which was required to work with the current HST data systems); and §5 discusses the performance of the compression method. Acronyms used in the paper are summarized in Table 1.

2. COMPRESSION REQUIREMENTS

There are many image compression methods available. For the ACS there were several considerations that made existing algorithms unsuitable:

- Only lossless compression methods are acceptable. Lossy image compression can give much better compression (to a few bits per pixel or less); however, because the ACS images have low noise and are subjected to very stringent quantitative analysis (e.g., to measure positions and brightnesses of faint objects), it was judged unacceptable to degrade the data through lossy compression.
- The computational requirements of the method must be very small. The ACS on-board computer is slow (a 16-MHz 80386) and has only enough buffer memory for a single WFC image. The algorithm needs to execute in a time not much longer than the CCD readout time (80 seconds for the entire image), and it must operate in a simple one-pass streaming mode (raw pixels in, compressed data out) that does not require additional memory.

Our preferred choice for ACS image compression was the Rice algorithm,[2] which is an excellent general-purpose compression algorithm. It is a lossless method, it works in a simple memory-less streaming mode, and it is quite fast. It is very effective at compressing the data too, typically producing compressed data volumes within a few percent of those produced by state-of-the-art algorithms that are far more computationally intensive and memory-hungry.

Unfortunately, the Rice algorithm was simply too slow to use on our processor. The optimized 80386 implementation was only capable of processing about 5×10^4 pixels/sec, meaning that it would required about 330 sec to compress a full WFC image. Since this is longer than the time required to read out an uncompressed WFC image to the spacecraft computer (256 sec), it did not meet our goal of speeding up the CCD readout.

Furthermore, the Rice algorithm was only marginally fast enough to compress one of the four WFC CCD readout streams "on-the-fly". There are two sets of readout electronics for each of the 2 WFC CCDs, so the camera is ordinarily read out in 4 data streams, each of which produces 5×10^4 pixels per second. At the total speed of 2×10^5 pixels/sec, it takes 80 seconds to read out the entire WFC image to internal buffer memory.

In the absence of compression, each of these 4 streams is written to a separate 8 Mbyte section of ACS buffer memory. The compression algorithm must be fast enough to compress at least one of these on-the-fly, i.e. during the readout process, so that the compressed data can be written directly into the ACS buffer. This is required because without compression the ACS WFC images completely fill the buffer memory, leaving no place to write the compressed data output.

We briefly considered the possibility of upgrading the ACS computer or using special-purpose Rice compression hardware in order to make Rice compression fast enough for the ACS application. Both options were rejected due to the cost of changing the ACS hardware design from the baseline design, which is based on the Space Telescope Imaging Spectrograph design. We concluded that a faster algorithm was needed if compression was to be used on-board ACS.

3. THE PIXEL-PAIR ALGORITHM

Analysis of the Rice implementation indicated that the slowest part of the algorithm is the portion that generates variable length codes for each pixel. The Rice algorithm, like most other compression methods, generates an output code for each input pixel. The input pixel requires 16 bits, while the output code may require 1, 2, 3, or more bits. Rarely, a hard-to-code pixel value may even require an output code longer than 16 bits.

These variable length codes do not respect the byte (8-bit) boundaries of the output data buffer. Thus there is extra logic required to insert (say) a 3 bit code when only 1 bit is left in the current output byte. This turned out to be the most time-consuming part of the Rice coding.

We decided that we needed a coding algorithm that produced codes matched to the 8-bit output buffer size. The output codes should therefore be 1 or 2 bytes long. But if we produce one output code for each 2-byte input pixel, this would produce a maximum compression ratio of only a factor of 2, which is not very good.

The key idea was to compress pixels in *pairs* rather than a single pixel at a time. Then the maximum compression ratio is a factor of 4 (1 byte out for 4 bytes in), which is adequate for our application. Compressing pixel pairs has the added benefit that the computational effort required for coding occurs only once for every 2 pixels, effectively reducing the total time required for code generation.

Our final algorithm was constructed using many of the ideas from the Rice algorithm. A pseudo-code summary of the scheme is given in Fig. 1. Here are notes (labelled by the line numbers from the listing) explaining some details of the algorithm:

09–10 Like the Rice algorithm, the first step in processing the input pixels is to take the difference of neighboring pixels. This produces a distribution that is symmetrical and is peaked near zero because neighboring pixels have correlated values. Since the algorithm works with pairs of pixels, there are two difference values (`d1` and `d2`).

11–16 Also like the Rice algorithm, the difference values are next mapped to the non-negative integers. Zero is mapped to zero, positive values are mapped to the even integers, and negative values are mapped to the odd integers (so difference values of -2, -1, 0, 1, 2 get mapped to 3, 1, 0, 2, 4.) The particular mathematical expression for mapping negative values (line **14**) assumes that the values are internally stored in twos-complement binary form (which is true for all modern computers of which we are aware.)

17 Given the two non-negative values `d1` and `d2`, we want to compute a single index value that specifies both quantities. The particular mapping we have chosen is shown in Fig. 2. This indexing scheme was designed to give good results for the case where the difference values have an exponential probability distribution, $P(d) \propto d^{-\gamma}$. Such distributions are typical of the differences between adjacent pixels in images; the Rice algorithm is also optimized for this difference distribution.

18–27 The index value is coded using 1 or 2 bytes depending on its amplitude. For small values less than `Threshold1`, which result from the common case of small differences between adjacent pixels, only a single byte is used. For somewhat larger values less than `Threshold2`, two bytes are used. For still larger values, we give up on the scheme of coding the 2 differences together and simply add the raw pixel values for each pixel in the pair (before taking any differences) to the output.

On decoding we recognize whether a 1-byte or 2-byte code was used by testing the first byte. If it is less than `Threshold1` then only a single byte was required; if it is greater than `Threshold1` then 2 bytes were used

```
Threshold1 = 224                    // Adjustable parameter (< 255) for the coding scheme
Offset = 255 * Threshold1
Threshold2 = 65536 - Offset
// A 1-byte flag precedes each 16-pixel (8-pair) block
FlagBit = 128
FlagVal = 0
FlagPtr = 0
BufPtr = 1
For i = 0 to NPixels-1 by 2 {
  // Compute differences between adjacent pixels
  d1 = p[i] - p[i-1]                // Note this must be changed for first pixel
  d2 = p[i+1] - p[i]
  // Map differences to non-negative values: 0 -> 0, positive d -> 2d, negative d -> -2d-1
  if (d1>=0) {
     d1 = d1<<1                     // '<<' is the left shift operator
  } else {
     d1 = ~(d1<<1)                  // '~' is the unary not operator, flips all bits
  }
  if (d2>=0) { d2 = d2<<1 } else { d2 = ~(d2<<1) }
  // Compute triangular index into d1, d2 values
  Index = d2 + (d1+d2)*(d1+d2+1)/2

  // Write 1, 2 or 4 bytes for each pixel pair
  if (Index < Threshold1) {
     Write Index to Output[BufPtr] as 1-byte value
     BufPtr = BufPtr+1
  } else if (Index < Threshold2) {
     Index = Index + Offset
     Write Index to Output[BufPtr..BufPtr+1] as 2-byte value
     BufPtr = BufPtr+2
  } else {
     Write original pixel values p[i], p[i+1] as
         two 2-byte values to Output[BufPtr..BufPtr+3]
     BufPtr = BufPtr+4
     // Set flag bit to 1 to indicate directly coded pixel values
     FlagVal = FlagVal | FlagBit
  }
  FlagBit = FlagBit>>1              // '>>' is the right shift operator
  if (FlagBit == 0) {
     // Write flag value to output buffer after finishing a block of pixels
     Output[FlagPtr] = FlagVal
     FlagBit = 128
     FlagVal = 0
     FlagPtr = BufPtr
     BufPtr = BufPtr+1
  }
}
```

Figure 1. Pseudo-code for the pair-coding algorithm. Assumes that the block size, `NPixels`, is a multiple of 16.

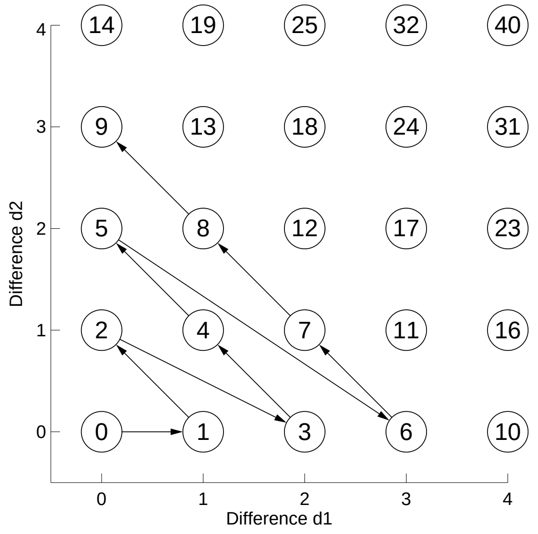

Figure 2. How a pair of non-negative values (d_1,d_2) is mapped to a unique index in the pair-coding algorithm.

and the second byte is read from the code stream. The values of `Offset` and `Threshold2` are computed from `Threshold1` (lines **02**–**03**) in such a way that we guarantee that the first byte of a 2-byte code will be in the range `Threshold1` to 255.

The variable `Threshold1` is a parameter of the coding scheme that determines when the algorithm switches between 1-byte codes and 2-byte codes for the pixel pair. It can be changed in the ACS implementation, but the standard value (224) is likely to suffice.

4–**6, 28**–**36** The strategy described above for recognizing 1-byte or 2-byte codes during decoding fails for the case where the raw pixel values are coded directly, because the first byte of a raw pixel value could have any value. We handle this case by including a single flag byte at the beginning of each block of 16 pixels (8 pixel-pairs.) The flag byte is initialized to zero. If a pixel-pair is directly coded, the corresponding bit in the flag byte is set to 1. Thus the flag bit is used during decoding to determine whether direct coding was used.

The best-case performance for this compression scheme is 9 bytes output (the flag byte plus 1 byte for each pixel pair) for each set of 16 pixels (32 bytes) input. The resulting compression ratio is $32/9 = 3.56$, and the code rate is 4.5 bits/pixel. The worst-case performance occurs when all pixel pairs are coded directly, in which case 16 pixels input get expanded by 3.1% to 33 bytes on output.

4. ACS IMPLEMENTATION

The data compression algorithm described in the previous sections has been implemented in C for the Intel 80386 microprocessor, for use in compressing WFC CCD images within the ACS. This implementation of the data compression algorithm will be referred to as the 'data compression function', to distinguish it from the abstract data compression algorithm which could be instantiated in many different languages running on many different computers. The data compression function described here can compress approximately 150,000 pixels per second, depending on the nature of the data.

The standard readout scenario for the WFC is to digitize the CCD data using four A/D converters, one for each quadrant of the detector, with the digitized data flowing simultaneously into four FIFOs. Each FIFO can contain 4096 digitized pixels. When the FIFOs simultaneously become half-full, an interrupt is generated to the 80386, whose flight software (FSW) then empties half-a-FIFO (2048 pixels) one FIFO at a time into a 32M byte buffer memory. Each FIFO ultimately contributes 8M bytes of uncompressed data per WFC readout. Given the rate of A/D conversion of approximately 20 microseconds per pixel, the data compression function can compress the output of only one of the four FIFOs on-the-fly, while the WFC readout is underway. At each FIFO half-full interrupt, a block of 2048 pixels are compressed, and the compressed data is stored in the buffer memory. The data from the

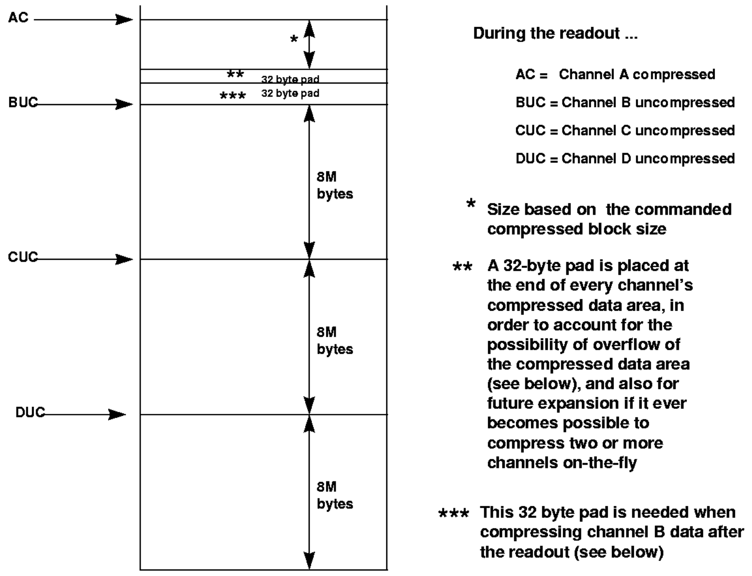
AC
BUC
CUC
DUC
*
**
32 byte pad
***
32 byte pad
8M bytes
8M bytes
8M bytes
During the readout ...
AC = Channel A compressed
BUC = Channel B uncompressed
CUC = Channel C uncompressed
DUC = Channel D uncompressed
* Size based on the commanded compressed block size
** A 32-byte pad is placed at the end of every channel's compressed data area, in order to account for the possibility of overflow of the compressed data area (see below), and also for future expansion if it ever becomes possible to compress two or more channels on-the-fly
*** This 32 byte pad is needed when compressing channel B data after the readout (see below)

Figure 3. Layout of the buffer memory while the WFC readout is in progress. The four FIFOs, or channels, are named A, B, C, and D; FIFO A is the one which is compressed on-the-fly.

other three FIFOs are stored uncompressed in the buffer memory (see Fig. 3), and are then compressed (almost) in place after the entire readout has been completed

In ground-based data compression, the amount of memory available to hold compressed data is typically not an issue. The amount of memory needed to hold compressed data is dependent on the data itself, and there is usually as much memory available as is needed. But in the ACS environment aboard the HST, the amount of buffer memory is a constrained resource. In addition, when data are dumped from the HST to the HST ground-system, the amount of data to be dumped must be specified in advance. In order to meet these constraints, the amount of buffer memory which will hold the results of compressing 2048 pixels (half-a-FIFO) is estimated by the HST ground-system and uplinked to the ACS FSW for each WFC readout; this is the "commanded compressed block size", annotated by the single asterisk * in Fig. 3. During the compression of any 2048 pixel input block, if the compressed data can fit within the commanded compressed block size, then no data are lost, and the remainder of the block is undefined (it contains whatever was previously stored in that location.)

However, if the compressed data for a 2048 pixel input block cannot fit within the commanded compressed block size, then as much compressed data as possible is written into the compressed block, but after that, data are lost. In order to make the data compression function as efficient as possible, checking for the overflow of a compressed block occurs at the end of compressing a block of each 16 pixels (rather than checking after compressing each pixel-pair). Thus, it is possible that a given compressed block can overflow by as much as 32 bytes; however the overflow will be overwritten when compressed data are written into the next block. The compression of each 2048 pixel input block is handled independently, so data can be lost while compressing some input blocks but not others.

During the compression process, the number of compressed blocks which have overflowed are counted, and the final count is included together with the compressed data when they are downlinked. During the decompression process, blocks which overflowed during the compression process can be identified by noting when the end of the compressed block is reached before the amount of reconstructed uncompressed data reaches 2048 pixels.

After the entire readout has been completed, the data from the channels B, C, and D in turn are compressed, each in an identical fashion to the compression of channel A. The same commanded compressed block size used to compress channel A is used during the compression of channels B, C, and D. As for channel A, the number of overflowing compressed blocks is counted and included with the downlinked compressed data.

Fig. 4 shows the details of compressing block B. Note how the compression begins in utilizing the 2nd of the 32-byte pads which follow channel A's compressed data. The upper bound on the commanded compressed block size is 2047 16-bit words. Thus, the compressed data for each block of 2048 input (16-bit) pixels must compress into at

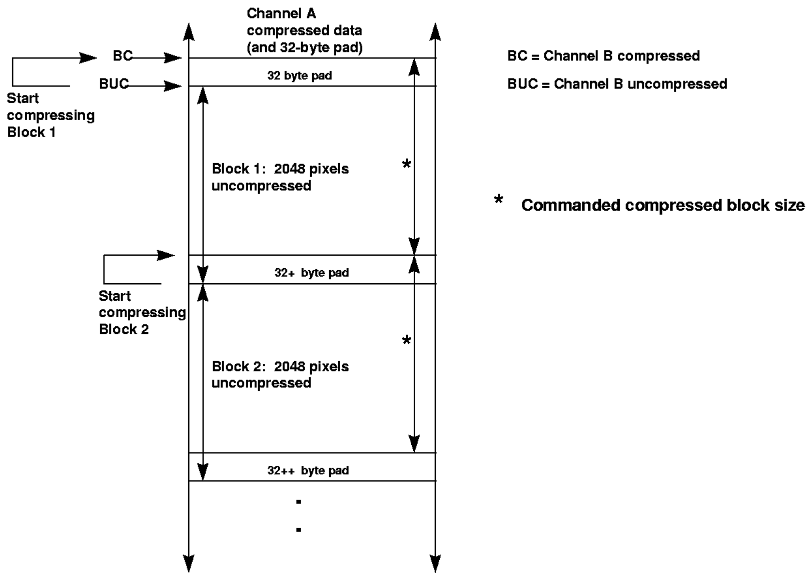
Channel A compressed data (and 32-byte pad)
BC
BUC
Start compressing Block 1
32 byte pad
Block 1: 2048 pixels uncompressed
*
Start compressing Block 2
32+ byte pad
Block 2: 2048 pixels uncompressed
*
32++ byte pad
BC = Channel B compressed
BUC = Channel B uncompressed
* Commanded compressed block size

Figure 4. Compression of channel B after the readout has completed.

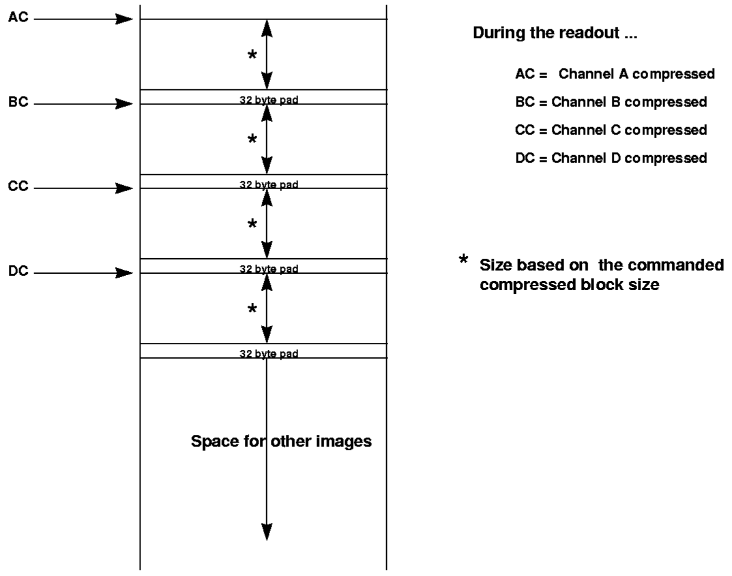
AC
BC
CC
DC
*
32 byte pad
Space for other images
During the readout ...
AC = Channel A compressed
BC = Channel B compressed
CC = Channel C compressed
DC = Channel D compressed
* Size based on the commanded compressed block size

Figure 5. Final memory layout after all 4 channels have been compressed.

most 2047 16-bit words before any overflow might begin for that block. However, as with channel A, the overflow is limited to 32-bytes; therefore in the worst case, the initial 32-byte pad will propagate down through the compressed blocks, gaining 1 word = 2 bytes per block. Therefore, the compressed data will never catch-up to and overwrite the uncompressed data.

At the end of channel B's compressed data, a pad of 32 bytes is inserted before channel C's compressed data blocks begin. Then, in similar fashion, blocks C and D are compressed. Fig. 5 shows the layout of buffer memory at the end of the compression process.

The implementation method described here is completely deterministic in the amount of buffer memory required. At the beginning of a WFC readout with compression, the buffer memory management flight software (designed and written by Robert Lampereur) calculates and allocates the required maximum amount of buffer to hold both the data from channel A compressed on-the-fly and the uncompressed data from channels B, C, and D. If insufficient

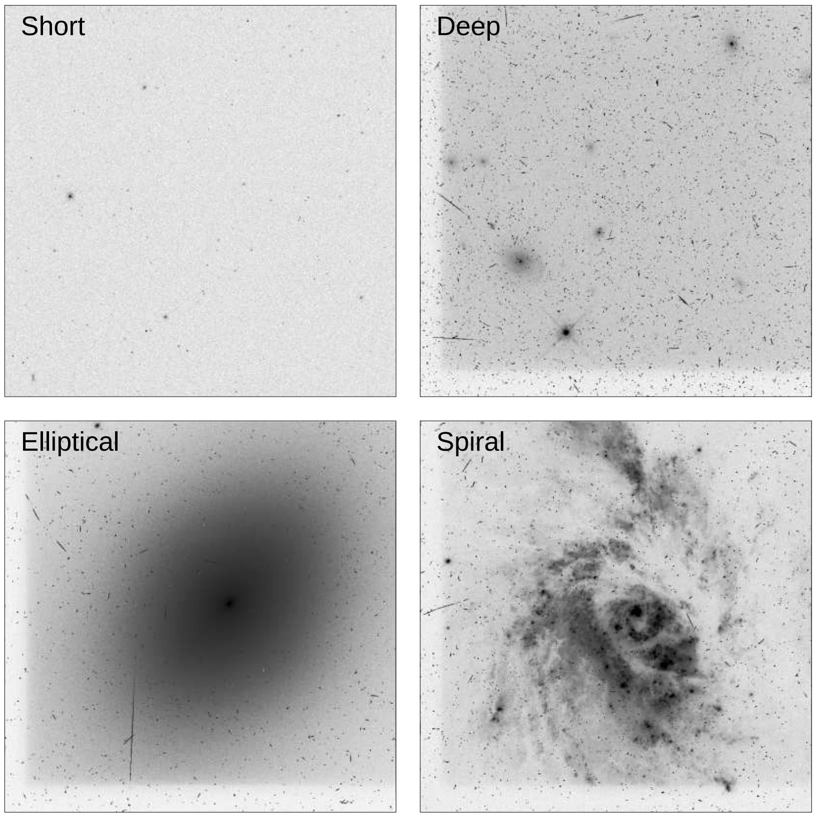

Figure 6. WFPC2 images used for compression performance tests.

buffer memory is available, an error message is generated and the readout will not take place. Otherwise, the readout begins, and the WFC readout software and then post-readout software calls the data compression function repeatedly to compress each 2048 pixel input block, passing in pointers to the beginning of the input block and to the beginning and end of the compressed output block. The data compression function returns an indicator to the caller saying whether or not the output block overflowed.

WFC subarrays can also be compressed. Recalling that overscan pixels have not been included in this discussion (until now), it may occur that the final input pixel block for a given channel contains fewer than 2048 pixels. In this case, the input block is padded-out with the value of the last pixel in the block up to 2048 pixels, and then the data compression function is called in the usual manner to compress the block.

5. PERFORMANCE OF ALGORITHM

We have tested the performance of the pair-coding algorithm by compressing a sample of images taken with the HST WFPC2 (Fig. 6). These images are suitable for testing because they show the typical characteristics that will be seen in ACS images, including a variety of astronomical objects (stars, galaxies, etc.), cosmic rays, and various CCD defects (readout noise, hot pixels, etc.) The WFPC2 does not have as large a dynamic range as ACS, but this has relatively little effect on compression because few pixels in astronomical images approach the full-well depth of the CCD.

The compression performance of the Rice algorithm, the pair-coding algorithm, and the familiar Gzip algorithm (which is far slower than the other methods) on the test images is shown in Table 2. The compressibility of the images varies over a rather wide range: the short exposure image has the lowest level of noise (it is dominated by CCD readout noise) and so is easiest to compress, while the deep, long-exposure image has the highest noise levels and is most difficult to compress. The ACS pair-coding method is clearly not competitive with the Rice algorithm on highly compressible images because of the 4.5 bit/pixel minimum size for pair-coding. (Note that the actual compressed size

Table 2. Compression for Test Images

	Compression (bits/pixel)		
Image	*Rice*	*Gzip*	*Pair*
Short exposure	2.762	2.909	4.542
Deep exposure	5.050	4.556	6.401
Big elliptical galaxy	4.247	5.224	5.317
Big spiral galaxy	4.247	4.690	5.500

for the short exposure image is very close to this limit.) For the other images, though, the pair-coding performance is not bad: the pair-coded images are 25–30% larger than the Rice-compressed images.

The poorer performance of the pair-coding method is a result not of coding pixels in pairs, but rather of forcing the code values to be multiples of 8 bits long. It would be straightforward to construct a version of the pair-coding algorithm that uses variable length codes and that would have performance comparable to (or perhaps better than) the Rice algorithm. Such a scheme would give better performance but would be computationally more demanding, making it unsuitable for ACS on-board compression; consequently we have not pursued this possibility.

REFERENCES

1. H. C. Ford and the ACS Science Team, "The HST Advanced Camera for Surveys," (this volume), 1998.
2. R. F. Rice, P.-S. Yeh, and W. H. Miller, "Algorithms for high speed universal noiseless coding," in *Proceedings of the AIAA Computing in Aerospace 9 Conference*, 1993.